OPT++: An Object-Oriented Toolkit for Nonlinear Optimization

J. C. Meza,R.A. Oliva[*], P.D. Hough, P.J. Williams[†]

December 9, 2003

Abstract

Object-oriented programming is a popular way of developing new software. The promise of this new programming paradigm is that software developed through these concepts will be more reliable and easier to re-use, thereby decreasing the time and cost of the software development cycle. This report describes the development of a C++ class library for nonlinear optimization. Using object-oriented techniques, this new library was designed so that the interface is easy to use while being general enough so that new optimization algorithms can be added easily to the existing framework.

1 Introduction

Object-oriented programming (OOP) is a popular way of developing new software. Unlike procedural programming, which emphasizes the development of algorithms to accomplish a specific task, object-oriented programming relies on the implementation of new data types called objects. The promise behind this programming paradigm is that software developed through these concepts will be more reliable and easier to re-use in new applications, thereby decreasing the time and cost of the software development cycle.

In this paper, we present the results of an ongoing project to develop an object-oriented toolkit for nonlinear optimization. This toolkit was first proposed by Meza [14] in 1992 for unconstrained optimization problems. Since that time, there have been many other researchers that have contributed to the development and new capabilities have been added over the years, including new parallel optimization methods[11],

[*]Computational Research Division, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley CA 94720. This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

[†]Computational Science and Mathematics Research, Sandia National Laboratories

parallel finite-difference gradient calculations[12], constrained optimization, and new testing capabilities. In addition, the toolkit has been used to solve many problems including molecular conformation problems, optimal control of chemical vapor deposition furnaces, and a parameter identification for an extreme ultraviolet lithography lamp model [16, 18, 15].

Many software packages have been developed to address various types of optimization problems. For an excellent overview of the available optimization software see for example [20]. Given the large number of optimization codes available, it is not surprising that it is sometimes difficult for the user to choose a good algorithm for a particular problem. This is especially true for the novice practitioner of optimization. In addition, even if the methods are inherently similar, the interface to the codes can be quite different making it difficult to experiment with various methods. To resolve some of these issues code designers usually resort to one of two tricks: 1) force the user to use a particular calling sequence or 2) the optimization codes are written using reverse communication.

Neither solution is very satisfying. If the optimization algorithm requires a particular calling sequence the user is forced into writing a subroutine that will interface between the optimizer and the function evaluator. While this is usually a straightforward task it may prove to be unwieldy and costly in certain situations. In particular, we would like to focus on cases where the function evaluator is described by the output of a simulation such as a finite-element analysis. For these simulation-based optimization problems, the prescribed interface may not be general enough to encompass all of the parameters required to do a simulation or it may require the user to package any extra information in a pre-defined format.

In the second option, called reverse communication, the optimization algorithm returns to the calling routine whenever it needs information to proceed. This information may be a function value, a derivative, or any other data that is required by the optimization algorithm. From the point of view of the user this is a better solution in that it requires less coding. From the point of view of the software developer however, the job is more difficult. Outside of the fact that this type of coding violates several good programming practices (for example, single entry-single exit codes), the code is also more difficult to debug.

To overcome some of these difficulties, several attempts have been made at designing optimization classes. In [26] Schoenberg developed a set of classes for the unconstrained optimization of arbitrary functions. Schoenberg describes three classes that together choose a particular algorithm, set the tolerances, and perform the actual optimization. Nichols et al. [21] have also developed optimization classes for linear operators in the physical sciences and specifically for linear operators arising from geophysical inversion problems. The Tookit for Advanced Optimization [28] (TAO) is another attempt at developing an optimization toolkit in an object-oriented framework. Another example is the Hilbert Class Libray [9] (HCL) designed to bridge the gap between off-the-shelf optimization software and application software that attempts to use it. More recently, the Object-Oriented Quadratic Programming [23] (OOQP) software package was described for the solution of convex quadratic programming problems.

The goal of this work is to use the ideas of object-oriented programming to overcome the obstacles mentioned above. In particular, we address the following issues: 1) better program interfaces for the user of optimization codes, 2) rapid evaluation of several optimization codes for a given problem, 3) rapid prototyping of new optimization algorithms, and 4) more re-usability of optimization components and codes.

The rest of this paper is organized as follows. In Section 2 we describe the mathematical problems that OPT++ is intended to address. Section 3 describes the software issues including: a short overview of object-oriented programming, the major classes within OPT++, the parallelization of OPT++, and the testing infrastructure developed. In Section 4 we give several examples of using OPT++ for some applications. We conclude in Section 5 with a discussion of future work.

2 Mathematical Formulation

The problem that we are interested in solving is that of a nonlinearly constrained optimization problem. The general form of this problem is given by:

$$\begin{aligned} \min_{x \in R^n} \quad & f(x) && && (1)\\ \text{subject to} \quad & h_i(x) = 0, && i = 1, \ldots, p, \\ & g_i(x) \geq 0, && i = 1, \ldots, m. \end{aligned}$$

where $x \in \Re^n$, and $f : \Re^n \to \Re$.

Here, both the objective function $f(x)$ and the constraint functions $h_i(x)$ and $g_i(x)$ are assumed to be general nonlinear functions.

For the case where there are no constraints present, there are many computational techniques for solving problem (1). Most of these techniques can be broadly classified as either using derivative information or being derivative-free. Under the category of methods that use derivative information are conjugate gradient methods, quasi-Newton methods, and Newton methods. Derivative-free methods include direct search methods, genetic algorithms, and simulated annealing.

In the context of constrained optimization, there are many variations of the methods mentioned above. In particular, there has been considerable interest recently in interior point methods. These methods attempt to solve problem 1 while maintaining strict feasibility throughout the optimization phase. These types of methods are particularly attractive for simulation-based optimization methods where many of the constraints must be satisfied at all times since infeasible points usually correspond to input parameters that are not valid for the simulation.

We have chosen to implement a particular variant known as a primal-dual interior-point method proposed by El-Bakry, Tapia, Tsuchiya, and Zhang []. The basic algorithm can be interpreted as damped Newton method on the perturbed Karush-Kuhn-Tucker (KKT) conditions (using the slack variable formulation),

$$F_\mu(x,y,z) = \begin{pmatrix} \nabla f(x) + \nabla h(x) y - \nabla g(x) w \\ w - z \\ h(x) \\ g(x) - s \\ ZSe - \mu e \end{pmatrix} = 0, \tag{2}$$

where $(s, w, z) \geq 0$, and μ is a perturbation parameter. El-Bakry, Tapia, Tsuchiya, and Zhang demonstrated that local and fast convergence of Newton's method can be retained by clever choices of perturbation and damping strategies.

3 Software Description

3.1 Object-Oriented Programming

The main concept behind object-oriented programming is data abstraction, which is the separation of the data and the procedures for manipulating that data from an application program. In many ways this is no different than good programming practices that try to keep the unnecessary details of a particular code from an end-user. The major difference in object-oriented programming is the ability to create user-defined data types and add them to an existing language thereby facilitating data abstraction. It is these new objects that give object-oriented programming its name. Through these new objects a computer language can be easily extended to handle new applications. A good example of this feature is the matrix package developed by Davies [5]. With this package, a user can define vectors and matrices as part of the language as well as use the standard operations defined for these objects, such as matrix addition, matrix multiplication, and inversion.

There are four main ideas that we will use from object-oriented programming: 1) abstraction, 2) classes and objects, 3) inheritance, and 4) polymorphism. We do not seek to give a full description of object-oriented programming, but merely to provide enough background material to discuss the new optimization classes. For a fuller description of object-oriented programming see [2, 10, 27].

The idea of abstraction in software design is an old one. In its most general form, abstraction means the ability to isolate information pertaining to a particular software design. In procedural programming for example, the idea of abstraction has led to the concept of modular programming. In object-oriented programming this idea is taken further through the introduction of abstract data types. For the purposes of this paper we will define an *abstract data type* as a user-defined extension to an existing language type. It will usually consist of a set of data structures and a collection of operations that can manipulate those data structures. Through the use of abstraction, code will hopefully be more robust since details of data structures and the algorithms that manipulate them are isolated from the user.

The next concept that is useful is that of a class. A *class* is a user-defined data type that allows for data hiding. A class typically consists of both a data structure and a group of subroutines that can manipulate these data structures. The data inside the structure is hidden from the user in that the only way to access it is through the subroutines defined as part of the class. In this manner, the user does not need to know about the particular implementation of the class but can concentrate on the use of it. An *object* is then just a particular instance of a class. The analogy in a procedural language is that of a variable being a particular instance of a pre-defined type such as an integer.

An overworked but simple example is that of a complex data type. In this example, we could define a class called **complex** that consists of a pair of existing language types, for example, two **floats**. A better example is that of a class called **Vector** that could be defined as an array of **floats** together with an **int** that defines the size of the vector. The difference between the class **Vector** and an array which already exists in most languages is that we can now define operations that can be used with these objects. Thus we could define vector addition using the standard "+" operator between two **Vectors** of the same size.

Inheritance allows for easy extension of capabilities and is perhaps the most important new concept after that of the class. The idea behind inheritance is that a new class can be defined using a previously defined class as a template. In the terminology of OOP the template is called the *base class* and the new class is *derived* from the base class by adding new features to it. One of the advantages of inheritance is that all of the algorithms defined as part of the old class are still valid for the new class. This results in more reusable code since it is not necessary to rewrite this portion of the algorithm for the derived classes.

The last concept we will discuss is called *polymorphism.* In C++, it is possible to have a pointer to a function that will perform different actions depending on what class it belongs to. In this way, it is possible to defer an algorithmic design decision until it is required. In the OOP terminology, these functions are called *virtual functions.* If a class contains virtual functions then it is called an *abstract class.* The reason for this distinction is that an abstract class can never be used to create an object, it can only be used as a base class for other derived classes.

We will take a slightly different approach by making a distinction between nonlinear problems and the methods used to solve these problems. The rationale for this decision is that users seldom are aware of the intricacies of the various methods nor should they need to become experts in numerical analysis. On the other extreme, the developer of optimization algorithms usually does not care about the details of how a problem is defined other than to know certain mathematical properties and some general problem characteristics. By making a distinction between problems and methods we can develop codes that will hopefully be used by both groups without having to rewrite the class libraries every time a new problem is presented or a new algorithm is developed.

The end-users of optimization algorithms are usually quite knowledgeable about the problems they are trying to solve. However, this information usually pertains to the physical problem or to the algorithmic details of the computer model. For instance, the user will know the dimension of the problem, whether analytic first or second derivatives are available, and a general idea about the cost of a function evaluation. The developer of optimization algorithms on the other hand, would usually like to know more about the mathematical properties of the problem as well as any special structure that might be exploited. For example, a developer might ask any or all of the following questions:

- How smooth is the function? Is the function C^0, C^1, C^2, etc.?
- Does the objective function have any special properties, for example, is it a linear function, a quadratic function, etc.?
- Is this a large dimensional problem?
- Is there any other special structure to the problem? For example, is this a partially separable problem?
- How many digits of accuracy does the objective function have? How many digits of accuracy does the derivative function have?
- Is the Hessian matrix sparse or dense?
- Is the objective function expensive to compute?

To consider the first property only, available optimization algorithms could be classified according to the amount of smoothness assumed in the objective function. For example, if the function is C^2 (twice continuously differentiable), then one could use a Newton method. However, if the function is only continuous, then one would probably use a direct-search method. For most users it may be difficult to prove how much continuity the objective function has and therefore they may not be able to pick the most appropriate method. What is more likely is that a user will use the first available optimization software or the easiest one to use among several, usually with mixed results.

It seems appropriate then to define nonlinear problems from the point of view of the user. On the other hand, optimization method classes should be defined from the point of view of the developer, as there is a great deal of similarity between various algorithms. In the rest of this section, we propose such a division and discuss a set of C++ classes for each one of these two cases.

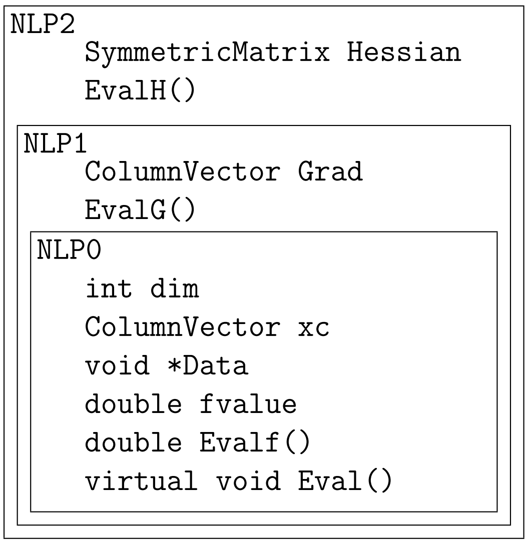


Figure 1: Nonlinear problem classes

3.2 Nonlinear Problem Classes

One of the first questions that arises is the degree of continuity in the objective function. This information may not be readily available, but what is clear is the availability of analytic derivatives. As such we've chosen to classify nonlinear programming problems by the availability of functions for computing the derivatives:

NLP0 – No derivative information available
NLP1 – Analytic first derivatives available
NLP2 – Analytic first and second derivatives available

In Figure 3.2, we present one implementation of a nonlinear problem class. The first class we define is called **NLP0** for NonLinear Problem C^0. This class contains information common to all problems including: 1) the problem dimension, 2) a current point, 3) a function value, and 4) a function to evaluate the objective function.

The class **NLP1** is derived from the base class **NLPO** by adding a member for the gradient and a function to evaluate the gradient. Likewise, the class **NLP2** is derived from **NLP1** by adding the necessary information to compute and store the Hessian. By using inheritance we can take advantage of the code that is already written at the lower levels.

We do not intend that these base classes cover every nonlinear problem. Instead, these classes should be viewed by the user as templates for new classes that contain the specific details of their own application problem. Since the optimization method classes described below will use the base classes, the optimization algorithms will still work with the new user classes without having to be rewritten.

In our implementation of the optimization classes, we have defined the functions that evaluate the objective function, gradient, and Hessian as virtual functions. As we mentioned in the previous section, this means that the **NLP0-2** classes are abstract classes and can only be used as base classes for other classes. This allows us to defer the definition of how the function, gradient, and Hessian are actually computed so that users can create their own definitions. In essence, the base classes contain placeholders for the codes that will be called to compute the objective function.

As part of the OPT++ implementation we provide 3 classes derived from **NLP0-2** called respectively **NLF0-2** that have a particular calling sequence to the required functions. These classes can be used to solve some simple optimization problems or can be used as templates for more sophisticated objective functions. In Section 4, we will give some examples using the **NLF** classes to demonstrate some of the features of our class libraries.

Finally, there are situations where a user may want to use first-order information without supplying analytic gradients. In this situation, OPT++ can automatically provide derivatives through a finite-difference calculation. This case is handled separately through the **FDNLF1** class that has been derived from the **NLP0** class.

3.3 Optimization Method Classes

There are many classifications possible for optimization algorithms, but most well-known methods can be grouped into one of three classes:

1. Direct Search methods - any method that does not require or use any first order information.

2. Conjugate gradient like methods - methods based on the conjugate gradient method.

3. Newton like methods - methods that use both derivative and second order information to build a quadratic model.

For example, methods such as the Nelder-Mead simplex method, the box method, the parallel direct search method, and pattern search methods fall into the direct search class. The nonlinear conjugate gradient method and limited memory BFGS methods fall into the Conjugate Gradient class. Finally the Newton class, could include methods such as finite-difference Newton, quasi-Newton methods, and inexact Newton methods. A simple taxonomy for some popular algorithms is given in Figure 3.3 as an example.

Based on this classification, we have implemented C++ classes for several methods including: 1) a Newton method, 2) a finite-difference Newton method, 3) a Quasi-Newton method, and 4) a nonlinear conjugate gradient method, 5) a parallel direct search method, 6) a generating search set method, and 7) a nonlinear interior point

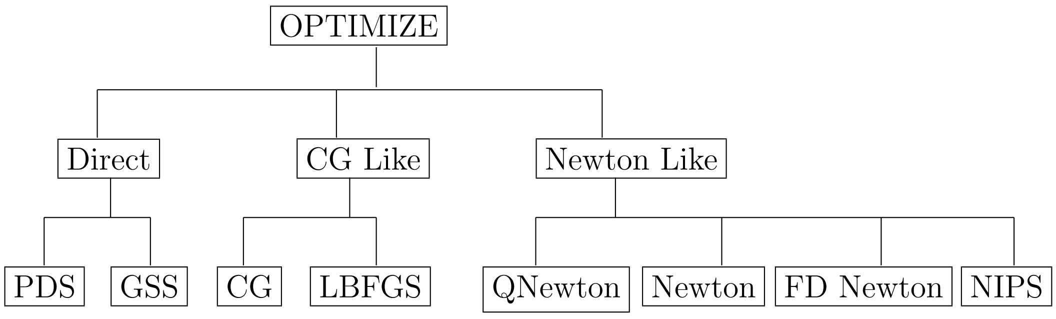


Figure 2: Optimization method hierarchy

method. In Figure 3.3, we present the class hierarchy for two of the implemented methods.

The base class, called **Optimize** consists of information that is required by all optimization classes. We note that once again we have used the concept of polymorphism through the use of the virtual function **optimize()**. This function is intended to be a placeholder for the actual function that will perform the optimization. Since each method class will have its own algorithm for computing the minimum of a function, it is not necessary to define it in the base class. However, it is important to define the interface at this point since it is common to all of the derived classes.

The next two classes **OptQNewtonLike** and **OptCGLike** are derived from the **Optimize** class. The major difference between these two classes is that the Newton-like classes require extra storage for the Hessian matrix. Finally, the last two classes **OptQNewton** and **OptCG** constitute the actual optimization methods. It is these two classes that define the optimization algorithms specific to each method. In the case of the **OptQNewton** class, the algorithm consists of a Quasi-Newton method with a BFGS update formula for the Hessian. The **OptCG** class implements a nonlinear conjugate gradient method.

As an example of the re-usability of object-oriented codes, all of the linear algebra is handled through the use of the matrix package developed by Davies [5], with some minor enhancements for the matrices that arise in the optimization algorithms. In addition, all of the optimization methods use the same line searches, a simple backtracking scheme and another one that is based on the algorithm by More and Thuente [19]

3.4 Constraints

Is the constrained optimization case a sub-class of the unconstrained optimization case or is a constrained optimization problem an unconstrained problem that hap-

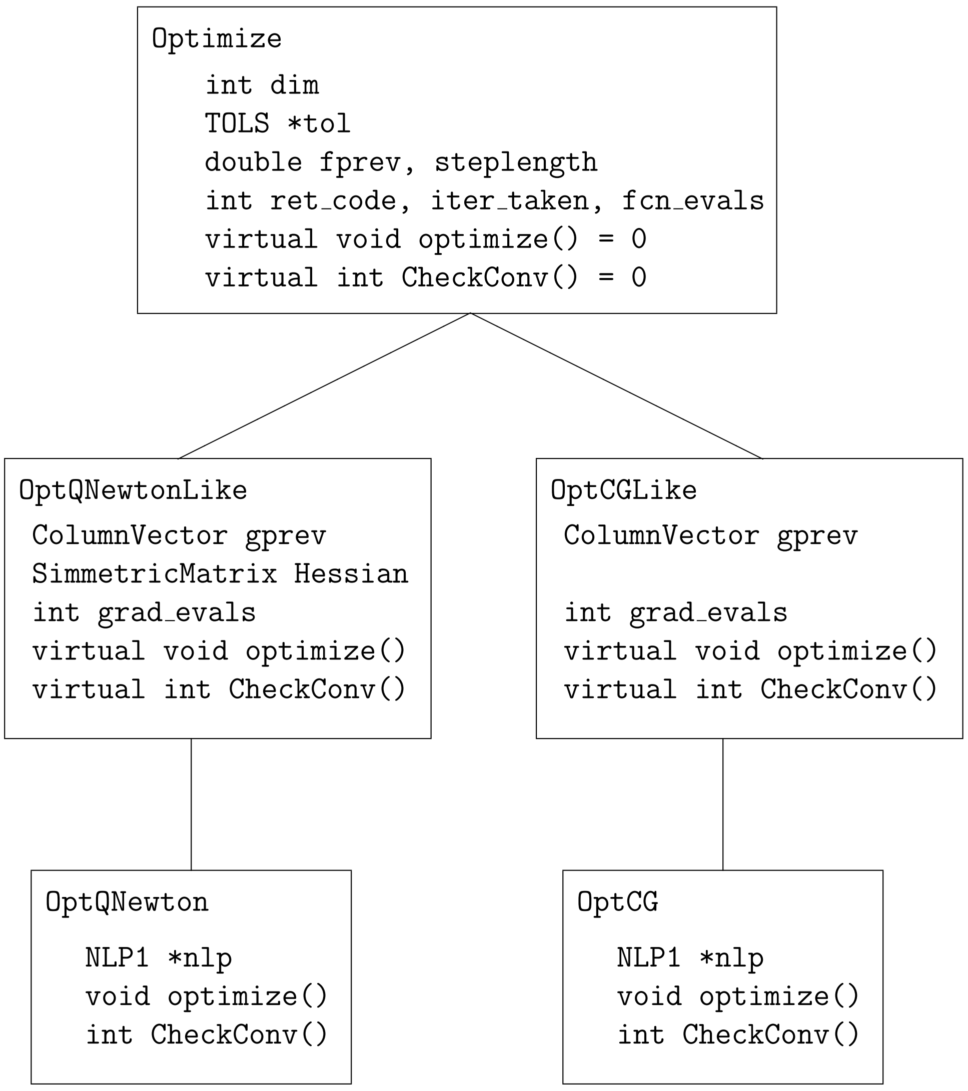


Figure 3: Optimization method classes

pens to have constraints? In the OOP terminology, this is the “inheritance” versus “composition” question, which has implications in the implementation of new classes. In composition or containment, class B is a member of class A. Membership is also known as “has-a” relationship.

We view a constrained nonlinear problem as an unconstrained problem that has constraints. We implement constraints in this manner to prevent code replication and to preserve the inheritance hierarchy in the nonlinear problem classes. From a code developer’s point of view, advantages to this approach are that we only have to add a pointer to a constraint object and constraint accessor methods to the existing nonlinear problem classes.

In Figure 4, we present the Constraint class hierarchy. From the ConstraintBase class, we derive **BoundConstraint, LinearConstraint**, **NonLinearConstraint** and **CompoundConstraint** classes.

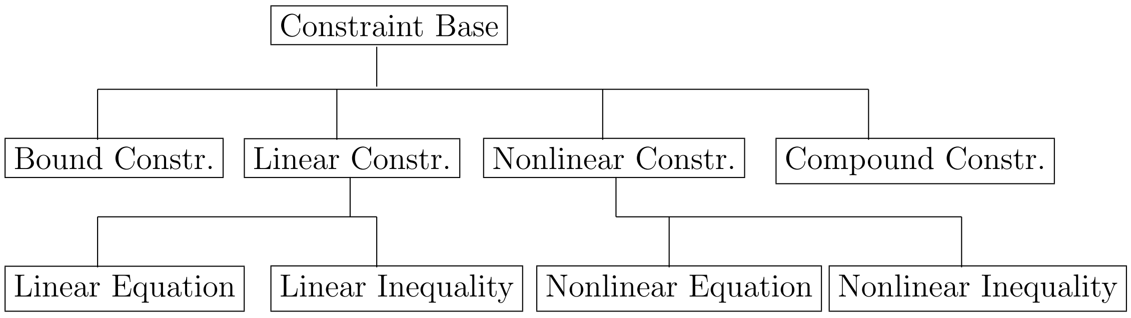
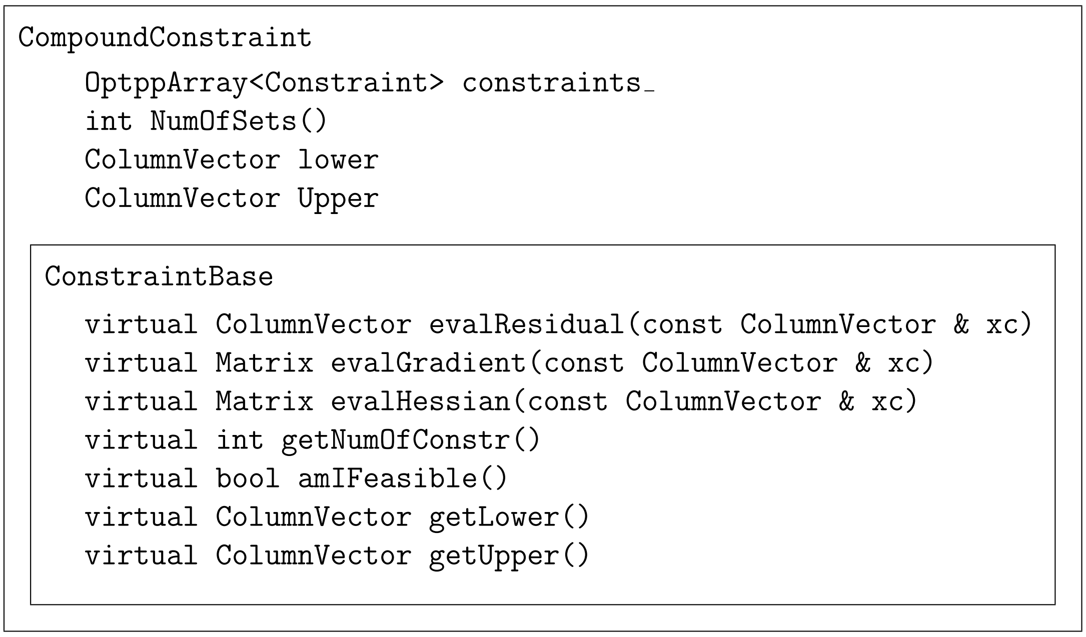


Figure 4: Constraint method hierarchy

- **BoundConstraint** – defines upper and lower bounds on real-valued variables;
- **LinearConstraint** – an abstract class, which provides common data and functionality to classes **LinearEquation** and **LinearInequality**;
- **NonlinearConstraint** – an abstract class, which provides common data and functionality to classes **NonLinearEquation** and **NonLinearInequality**.

The following pure virtual functions are declared in **ConstraintBase**: evalResidual, evalGradient, evalHessian, getUpper getLower, getConstraintValue, getNumOfCons, amIFeasible. A pure virtual function is a function that has not been defined in the base class; see [10]. To prevent compile-time errors, each derived class must provide a definition for the pure virtual function.

The class **LinearConstraint** is derived from **ConstraintBase** by adding members for the coefficient matrix A, the matrix-vector product Ax, lower and upper bounds, and a flag to determine whether the constraint is written in standard form. The derived classes of **LinearConstraint, LinearEquality** and **LinearInequality**,



```
CompoundConstraint
    OptppArray<Constraint> constraints_
    int NumOfSets()
    ColumnVector lower
    ColumnVector Upper

  ConstraintBase
    virtual ColumnVector evalResidual(const ColumnVector & xc)
    virtual Matrix evalGradient(const ColumnVector & xc)
    virtual Matrix evalHessian(const ColumnVector & xc)
    virtual int getNumOfConstr()
    virtual bool amIFeasible()
    virtual ColumnVector getLower()
    virtual ColumnVector getUpper()
```

Figure 5: Constraint object classes

differ in the implementation of evalResidual, evalGradient, evalHessian, and amIFeasible functions. Similarly, the **NonLinearConstraint** class is derived from **ConstraintBase** by adding a pointer to a **NLP** object. Like the objective function, the nonlinear constraint functions are classified according to the availability of derivative information. However, the argument list of the constructors for the two objects differ in the storage types for the function's value, gradient, and Hessian.

In OPT++, the standard form for a constrained nonlinear programming problem is

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & A_e x = b_e, \\ & A_i x \geq b_i, \\ & h(x) = 0, \\ & g(x) \geq 0, \\ & l \leq x \leq u, \end{array} \tag{3}$$

where $f(x), h_i(x), g_i(x)$ are nonlinear functions and A_e and A_i are real-valued linear coefficient matrices. A naive implementation of constrained optimization would lead to a combinatorial explosion of algorithms to solve for example, bound-constrained, linear equality constrained, nonlinear constrained, as well as bound and linear equality constrained problem formulations.

To simplify the use of mixed constraint sets in OPT++, we create a **CompoundConstraint** class. A **CompoundConstraint** is an array of heterogenous constraints. Construction of the **CompoundConstraint** class eliminates the need to implement different versions of optimization algorithms based on constraint type. The design of **CompoundConstraint** places the burden of properly managing the constraints on the algorithm developer. In this design, separate treatment of constraint classes is hidden from the user.

The class **CompoundConstraint** is derived from **ConstraintBase** by adding a member for an array of constraints, a counter of the number of constraint sets in the problem formulation, as well as comparison and insertion sort functions. Without loss of generality, inside the **CompoundConstraint** constructor, the constraints are sorted so that equality constraints are followed by inequality constraints. Why? Optimization algorithms may treat categories of constraints differently. If constraints are pre-sorted, the optimization algorithm does not have to continually query about constraint type.

Currently, OPT++ does not support sparse constraints. Therefore, a bound must be given for each constraint even if only a subset of the constraints have finite bounds. An infinite lower bound is specified by setting

$$l_i \leq -10^{10}.$$

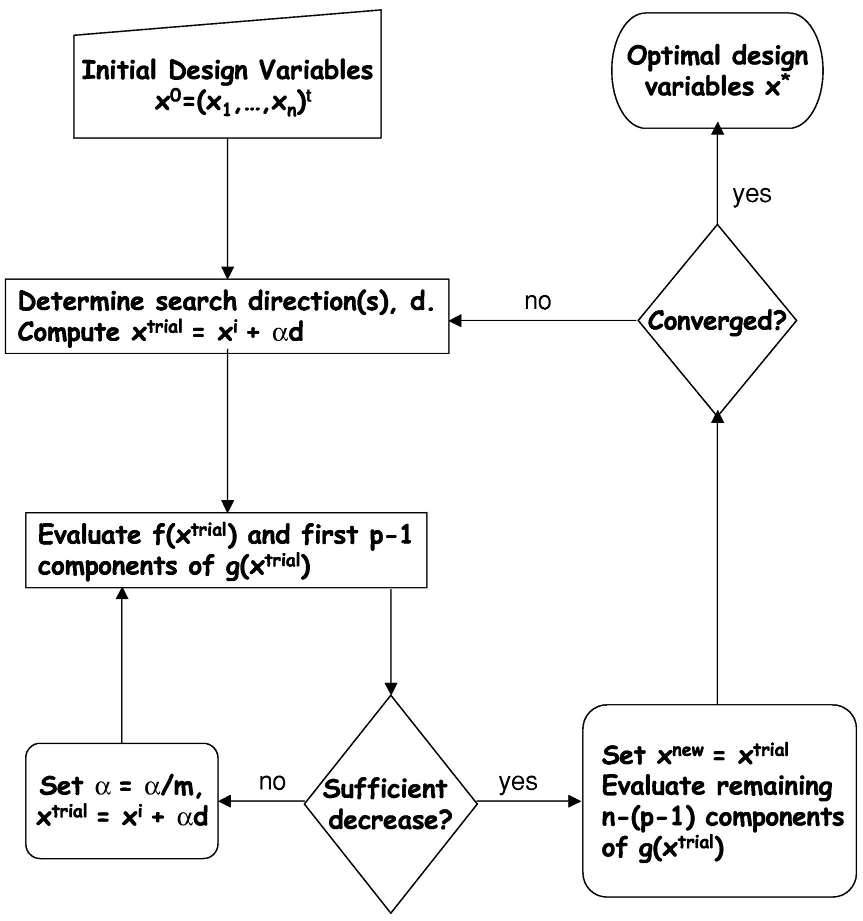
Similarly, an infinite upper bound is specified by

$$u_i \geq 10^{10}.$$

3.5 Parallel Optimization Methods

One approach to mitigating the high computational cost of the functions targeted by OPT++ algorithms is to take advantage of the availability of multiple processors. In order to enable this capability, parallel versions of some of the OPT++ algorithms have been implemented using MPI [7]. The parallelism in OPT++ currently takes the form of coarse-grained parallelism, and in particular, it consists of the ability to perform multiple function evaluations simultaneously. This is particularly advantageous when finite-difference gradient approximations are needed or when derivative-free methods are used, both of which are common in simulation-based optimization. Below we describe in more detail how the algorithms in OPT++ make use parallelism.

When an analytic gradient is not available, the gradient-based algorithms in OPT++ must approximate it using finite-difference computations. When each function evaluation requires the execution of an expensive simulation, the cost of solving the optimization problem can very quickly become prohibitive. One approach to addressing this problem was introduced by Byrd, Schnabel, and Shultz in 1988 [4]. They suggest a straightforward way to take advantage of multiple processors when using a line search method with finite-difference gradients. In particular, extra processors are used to compute components of the finite difference gradient at the trial point while the function is being evaluated at that point. This is referred to as a speculative gradient computation, and the idea applies equally well to any gradient-based algorithm. Figure 6 shows the flow of a generic gradient-based optimization algorithm with the speculative gradient modification. Once particularly nice feature of note is that while this approach clearly leads to substantial computational savings, there is no penalty when the trial point is not accepted.



Initial Design Variables
$x^0=(x_1,\ldots,x_n)^t$
Optimal design variables x^*
yes
Determine search direction(s), d.
Compute $x^{trial} = x^i + \alpha d$
no
Converged?
Evaluate $f(x^{trial})$ and first p-1 components of $g(x^{trial})$
Set $\alpha = \alpha/m$, $x^{trial} = x^i + \alpha d$
no
Sufficient decrease?
yes
Set $x^{new} = x^{trial}$
Evaluate remaining n-(p-1) components of $g(x^{trial})$

Figure 6: *While the function is being evaluated at the trial point, the remaining $p-1$ processors are used to calculate up to $p-1$ components of the finite difference gradient. If the trial point is accepted, we already have $p-1$ components of the gradient available and only need to calculate the remaining $n-(p-1)$ components, where n is the dimension of the problem. If the trial point is not accepted, we simply try again at the next trial point. Note that no time is lost because the function evaluation is required regardless.*

As an additional note, it is observed in [4] that it is possible to extend this idea to using any additional processors to compute as many components of a finite-difference Hessian as possible. In OPT++, however, we currently rely on other Hessian approximation (e.g. BFGS). We plan to incorporate speculative Hessian evaluations in the future.

Derivative-free optimization methods are also commonly used to solve simulation-based optimization problems. They have the obvious advantage of not requiring a gradient, but they are also robust when the accuracy of the function is low. One of the derivative-free methods implemented in OPT++ is the parallel direct search (PDS) algorithm of Dennis and Torczon [6]. Direct search can be briefly described as follows. Starting from an initial simplex, S_0, the function value at each of the vertices in S_0 is computed and the vertex corresponding to the lowest function value, v_0, is determined. Using an underlying grid structure, S_0 is reflected about v_0 and the function values at the vertices of this rotation simplex, S_r, are compared against the function value at v_0. If one of the vertices in S_r has a function value less than the function value corresponding to v_0, then an expansion step to form a new simplex, S_e, is attempted in which the size of S_r is expanded by some multiple, usually 2. The function values at the vertices of S_e are compared against the lowest function value found in S_r. If a lower function value is encountered, then S_e is accepted as the starting simplex for the next iteration; otherwise S_r is accepted for the next iteration. If no function value lower than the one corresponding to v_0 is found in S_r, then a contraction simplex is created by reducing the size of S_0 by some multiple, usually 1/2, and is accepted for the next iteration.

Because PDS only uses function comparisons, it is easy to implement and use. Furthermore the rotation, expansion, and contraction steps are all well determined, so it is possible to pre-compute a set of grid points corresponding to the vertices of the simplices constructed from various combinations of rotations, expansions, and contractions. An example of such a scenario is shown in Figure 7. It is at this point where the introduction of parallelism becomes obvious and advantageous. Given this set of grid points, called a search scheme, each vertex can be assigned to a different processor thus allowing the function values at multiple vertices to be computed simultaneously. Hence the "parallel" in PDS. Once the function has been evaluated at all of these vertices, the one corresponding to the lowest function value can be identified, and the algorithm can move on to the next set of vertices.

The final OPT++ algorithm that leverages the simultaneous function evaluation capability is Trust Region-Parallel Direct Search (TRPDS) algorithm developed by Hough and Meza [11]. TRPDS employs the standard trust-region framework, but uses PDS to solve a non-standard subproblem to compute the step at each iteration. An example iteration of the TRPDS algorithm is shown in Figure 8. Since TRPDS has both gradient-based and derivative-free stages in the algorithm, it makes use of simultaneous function evaluations in both of the manners described above. The minor exception is that TRPDS does not currently support a speculative gradient

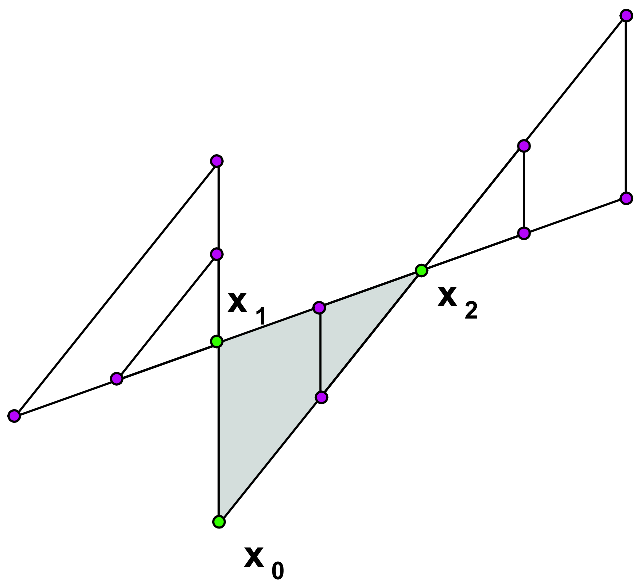


Figure 7: *This figure shows a set of vertices that have been pre-computed by considering various combinations of reflections, expanasions, and contractions around the current iterate. Only the function value is required at each point, and the evaluations are independent of each other, so each vertex can be assigned to a different processor.*

evaluation, though it does perform the finite-difference calculation in parallel. We plan to incorporate a speculative capability in the future.

We close this section with a few notes about our future plans. As mentioned earlier, the parallelism in OPT++ is currently coarse-grained parallelism in the form of simultaneous function evaluations. There are, however, other levels at which parallelism can be introduced. One that we intend to address is parallelism in the linear algebra. The problems that OPT++ has typically been used for in the past have had a small number of optimization parameters, so parallelism in the linear algebra has not been necessary. As OPT++ receives more use on large-scale problems, such as protein folding, it will become increasingly important to reduce the amount of time spent in the linear algebra. The other form of parallelism we intend to accommodate is parallelism within a single function evaluation. For simulation-based optimization problems, it is often the case that the simulation itself can be executed in parallel. While this can be done now through the clever use of system calls and function wrappers, it can be a tricky process with the onus on the user. We intend to incorporate this capability into OPT++ so that the user need only specify how much parallelism is required at each level.

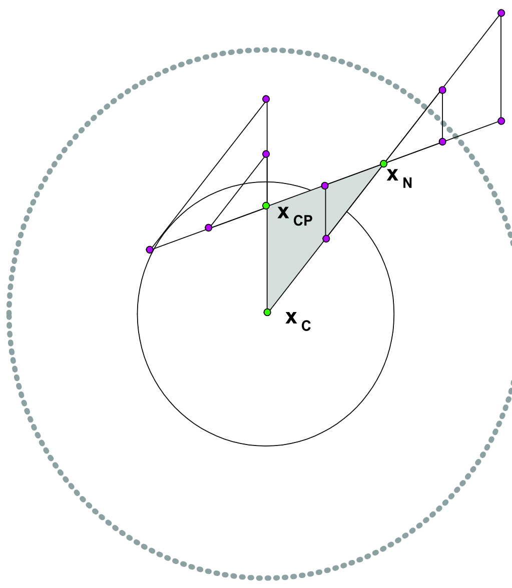


Figure 8: *Overview of the* TRPDS *algorithm. The point* $\mathbf{x}_{CP}$ *is the Cauchy point,* $\mathbf{x}_N$ *is the Newton point, and* $\mathbf{x}_C$ *is the current point. These points are used to initialize the simplex over which PDS approximately minimizes the function. The solid circle represents the trust region. The step length is allowed to be twice the size of the trust region (dotted circle) to allow for the possibility of taking a step longer than the Newton step.*

3.6 Regression Testing

Test problems are an important element of computational algorithm development, not only as a tool to assess implementation correctness, but also in performance comparison among competing algorithms.

OPT++ includes several regression tests that are run to ensure software reliability during the course of development. These include testing the optimizations algorithms using problems from the Hock-Schittkowski collection, problems from the MINPACK2 suite, as well as tests that are aimed at ensuring the integrity of the base components of OPT++, such as the constraint and NLP classes. These tests are run daily as part of an automatic regression testing script that compiles and archives the results and test logs. In the case of a failure of any of the tests an email is automatically generated and sent to the OPT++ software development team. Through this process, we have been able to detect bugs and correct them quickly and more efficiently.

Recently, all of the Schittkowski test problems for nonlinear programming were added into the testing environment of OPT++. This collection consists of over 300 problems designed to test various aspects of nonlinear optimization codes. The Schittkowki test problem collection was incorporated into OPT++ via an object- oriented library described in [13]. One of the featrures of this library is that it allows the development team to use this large collection of problems as a database. For example, one can query the test problems for specific attributes such as problem dimension, number of constraints, type of constraints (bound, inequality, equality), and availability of derivatives. This permits an efficient selection of the appropriate test problems for the algorithm being tested. The code in Figure 9 displays an example of a testing routine for the LBFGS algorithm by selecting all unconstrained problems having dimension larger than 10.

4 Applications

4.1 Example Code

To illustrate some of the concepts, we now present several examples that solve a small nonlinear optimization problem using the optimization classes. The first two test problems consist of Rosenbrock's function,

$$\min_x \; 100(x_2 - x_1^2)^2 \;+\; (1-x_1)^2.$$

In the first example, we assume that first derivatives are not available. In the second example, we asume that that first derivatives are available but that second derivatives are not available. For the solution method, we will use a quasi-Newqton method that employs a BFGS update formula for the Hessian.

Figures 10 - 11 display the source listings for the two sample problems. There are three major sections in the example code: 1) the function definition, 2) the nonlinear

```
int main() {
  int max_id = 309;
  for (int i=1, i< max_id; i++) {
    STPH tp(i);
    if (tp.getDim()>10 && !tp.hasConstraints()) {
    FDNLF1 nlp = tp.MakeFDNLF1();
    OptLBFGS solver(&nlp);
    solver.optimize();
    int rc = solver.getRetCode();
    if (rc > 0)
      cout << ''Test Problem '' << i << '' passed'' << endl;
    else
      cout << ''Test Problem '' << i << '' failed'' << endl;
  }
}
```

Figure 9: Sample testing code.

```
#include "NLF.h"
void rosenInit(int ndim, ColumnVector& x);
void rosen0Eval(int ndim, const ColumnVector& x,
               double& fx, int& result);

int main() {

  int ndim = 2; // problem dimesion

  FDNLF1 nlp( ndim, rosen0Eval, rosenInit );

  OptQNewton optobj(&nlp);

  optobj.setMaxIter(1000);

  optobj.optimize();

  optobj.printStatus("status");
}
```

Figure 10: Unconstrained example, no derivatives.

```
#include "NLF.h"
void rosenInit(int ndim, ColumnVector& x);
void rosen1Eval(int ndim, const ColumnVector& x,
               double& fx, ColumnVector& gx, int& result);

int main() {

  int ndim = 2; // problem dimesion

  NLF1 nlp( ndim, rosen1Eval, rosenInit );

  OptQNewton optobj(&nlp);

  optobj.setMaxIter(1000);

  optobj.optimize();

  optobj.printStatus("status");
}
```

Figure 11: Unconstrained example, function with first derivatives.

problem definition, and 3) the optimization method definition. In the first case, no analytic derivatives are supplied by the function, so we allow OPT++ to compute them via a finite-differnce approximation. This is encapsulated through the definition of the problem as a **FDNLF1**. In the second example (see Figure 11), analytic first derivatives are available, so we create an object of type **NLF1**. The corresponding definition of the objective function is also modified to reflect the availability of first derivatives. The two components needed to specify this object are the dimension of the problem and a pointer to a function. In both cases the last step consists of creating an optimization method object from the **OptQNewton** class. We then call the member function **optimize** to do the actual optimization. Finally the solution is printed using the **printStatus** member function.

We note that if the user would now like to try a different optimization method, the procedure would consist of replacing the creation of the **OptQNewton** object with a different method object, for example an **OptCG** object to try the nonlinear conjugate gradient method.

The third example consists of the following constrained optimization problem taken from the Hock and Schitkowski test set:

$$\begin{array}{rc} \min\limits_{x} & (x_1 - x_2^2)^2 \ + \ 1/9(x_1 + x_2 - 10)^2 \ + \ (x_3 - 5)^2, \\ \text{subject to} & -4.5 \le x_1 \le 4.5, \\ & -4.5 \le x_2 \le 4.5, \\ & -5.0 \le x_2 \le 5.0, \\ & x_1^2 + x_2^2 + x_3^2 \le 48. \end{array} \tag{4}$$

```
#include "NLF.h"
#include "OptNIPS.h"
void hs_ic(int ndim, ColumnVector& x);
void hs_f2(int ndim, const ColumnVector& x,
               double& fx, ColumnVector& gx, int& result);
void hs_inq1(int ndim, const ColumnVector& x,
               double& fx, ColumnVector& gx, int& result);

int main() {
   int n = 3;

  // Create Bound constraints
  ColumnVector lower(n), upper(n);
  lower << -4.5 << -4.5 << -5.0;
  upper << 4.5 << 4.5 << 5.0;
  Constraint boundc = new BoundConstraint(n,lower,upper);

  // Nonlinear Inequality constraint
  NLF1* inq_nlf = new NLF1(n,1, hs_inq1, hs_ic);
  NLP* inq_nlp = new NLP( inq_nlf );
  Constraint inqc = new NonLinearInequality(inq_nlp);

  // Compounded constraint object
  CompoundConstraint* constraints = new CompoundConstraint(inqc,boundc);

  // Construct non-linear problem with constraints
  NLF2 nips(n, hs_f2, hs_ic, constraints);

  // Build the NIPS optimization object
  OptNIPS objfcn(&nips, update_model);

  // Run optimization
  objfcn.optimize();

  // Write output
  objfcn.printStatus("Parameters");
  ostream* os = objfcn.getOutputFile();
  nips.fPrintState(os, "Solution from NIPS");

  objfcn.cleanup();
}
```

Figure 12: Constraints example.

4.2 Multi Material Heat Equation

The next example involves an inverse problem formulation of a problem in thermal analysis. The objective is to find the parameters of a model that would produce a known temperature field. This problem is representative of several problems that we have worked on, including optimal control of a chemical vapor deposition furnace [17, 18], and a problem in parameter identification for an extreme ultraviolet lithography lamp model.

For this problem, the objective function is defined as follows:

$$\min_x \; ||T(x) - T^*||_2,$$

where T^* is a prescribed temperature field and $T(x)$ is the solution to the heat equation

$$\frac{\partial T}{\partial t} - \kappa \nabla^2 T = 0$$

with appropriate boundary conditions over a specified domain. For this example the domain is the region $[0, 3] \times [0, 2]$, and each unit square within the domain represents a different material with the boundary condition (fluxes) as indicated in figure 13. We use a program that computes t he temperature on a regular mesh (of fixed size) by solving the heat equation given three input variables corresponding to the three fluxes. This temperature field is saved as $T(x)$. In a real-world scenario the prescribed temperature field T^* would originate from a physical experiment, but for the purposes of this example T^* was generated numerically using a specific set of inputs parameters x^*, namely $T^* = T(x^*)$.

We then attempted to recover x^* by solving (4.2). Since analytic derivatives were not available, we used two direct search methods and compared the solution to two gradient-based methods that used finite-difference gradients supplied by OPT++. The results are presented in Tables 4.2 - 4.2. For the direct search methods we used a step tolerance $= 10^{-5}$ and a function tolerance $= 10^{-4}$. For the gradient based methods we used a function tolerance $= 10^{-8}$, and a step size for the finite difference gradient $= 10^{-6}$.

The first direct search method used was PDS, the Parallel-Direct Search methods of section ??. The second was GSS, or Generating Set Search method, using three types of generating basis. In GSS2 we use the standard basis of size $2N$, namely the columns of $\{I, -I\}$ where I is the identity. In GSS1 we use the standard basis of size $N + 1$, namely $\{I, -\mathbf{1}\}$ where $\mathbf{1}$ is the vector with all entries equal to 1. In GSSa and GSSb we augument the standard $2N$ basis with search directions that couple the search directions in pairs and triplets, respectively.

To check the effects of the mesh size on the results we used meshes with two different divisions per unit length (dpu), namely 8 and 16. Both meshes produced consistents results, with the main differences being in the objective function value at the solution (this is expected as the objective function depends on the size of the mesh).

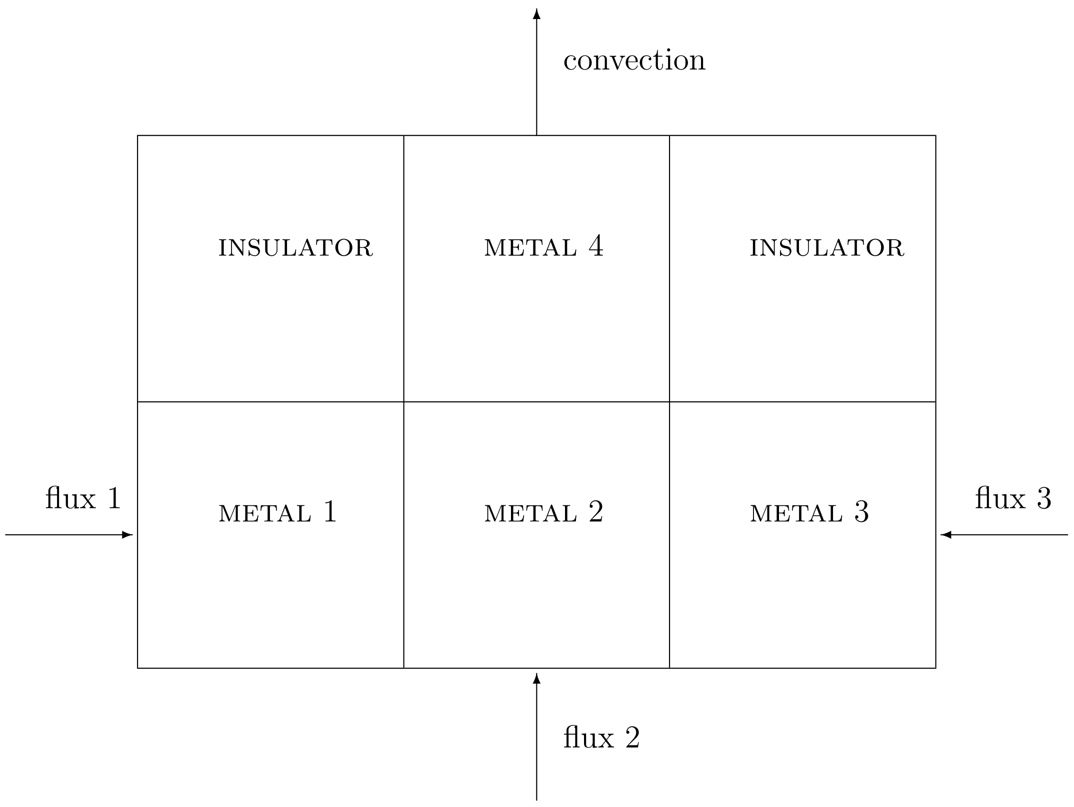


Figure 13: Multimaterial model problem.

Table 1: Comparison of direct search methods for the multimaterial problem.

	dpu	fevals	$f(x^*)$n	$\|\|s\|\|$
PDS	16	1092	1.7264e-3	1.5259e-5
GSS2	16	1168	1.5125e-3	8.1739e-6
GSS1	16	3125	6.2097e-3	7.9511e-6
GSSa	16	1077	5.1055e-4	6.5979e-6
PDS	8	1156	6.3770e-4	3.0518e-5
GSS2	8	1483	5.1255e-3	8.5457e-6
GSS1	8	3125	6.2097e-3	7.8511e-6
GSSa	8	1077	1.2042e-4	6.5979e-6
GSSb	8	595	3.8313e-4	9.0622e-6

Table 2: Comparison of gradient based methods for the multimaterial problem.

	dpu	fevals	$f(x^*)$	$\|\|g(x^*)\|\|$
QNewton	8	309	7.0871e-9	3.9285e-5
LBFGS	8	315	3.5252e-7	5.3536e-3
QNewton	16	287	1.7695e-8	7.3354e-5
LBFGS	16	623	2.2434e-7	2.8814e-3

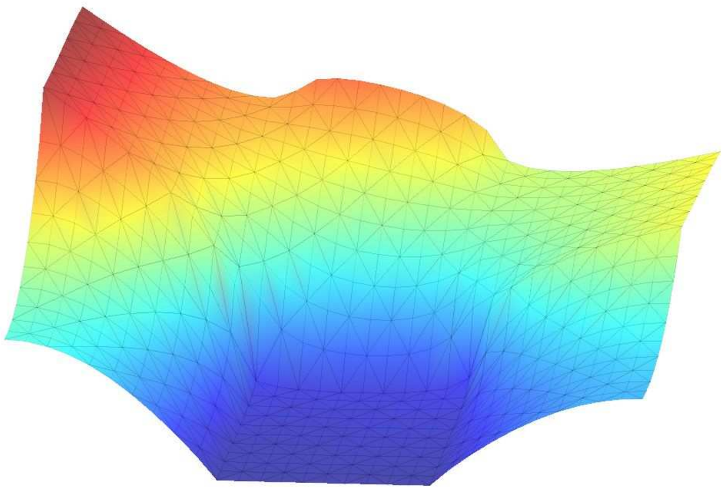


Figure 14: Target temperature profile for multimaterial problem.

4.3 Protein Folding

In the final example we use OPT++ to minimize the AMBER potential energy of a protein. AMBER is a prominent empirical energy model used in optimization approaches to the problem of predicting the spacial configuration of a protein given its sequence of amino acids, also know as the protein folding problem. All optimizations approaches to this problem assume that the natural configuration of the protein minimizes an energy potential, but the exact form of this energy is a current area of research.

The AMBER model considers the distances r_i between pairs of bonded atoms, the angles θ_i between consecutives bonds, and the dihedral angles ϕ_i defined by sequences of four bonded atoms, and assigns positive energies to the deviation from empirical values for these parameters. In addition to these bonded interactions, AMBER includes terms for the electric potential energy and Lennard-Jones interactions between pairs of non-bonded atoms. The form of the AMBER function can then be expressed as:

$$\begin{aligned} E_{AMBER} &= \sum_{i=1}^{N_\mathrm{b}} \bar{a}_i (r_i - \bar{r}_i)^2 + \sum_{i=1}^{N_\mathrm{a}} \bar{b}_j (\theta_i - \bar{\theta}_i)^2 + \sum_{i=1}^{N_\mathrm{d}} \bar{c}_i \cos(\bar{m}_i \phi_i - \bar{\gamma}_i) \\ &+ \sum_{i=1}^{N} \sum_{j=i}^{N} \left[\frac{\bar{q}_i \bar{q}_j}{r_{ij}^2} + \left(\frac{\bar{\sigma}_i \bar{\sigma}_j}{r_{ij}} \right)^6 + \left(\frac{\bar{\sigma}_i \bar{\sigma}_j}{r_{ij}} \right)^{12} \right] \end{aligned} \tag{5}$$

where the first three terms are the bonded interactions, with N_b the number of bonds, N_a the number of angles, and N_d the number of dihedral angles in the protein, and where the last summation incorporates the interactions among pairs (i, j) of non-bonded atoms, with r_{ij} denoting the distance between them. Quantities in (5) marked with over-bar are predefined constants dependent on the specific atoms involved in the interaction.

In this problem, a protein with N atoms is represented as a vector of length $3N$ corresponding to the atom's cartesian coordinates. Hence, typical problem sizes are in the thousands, as real proteins are made up of hundreds of amino acids, each with a dozen or so atoms. For some optimization methods, the size of this problem can create difficulties, especially if they either store or solve a size N linear system at each iteration. One method that is more adequate for large-size unconstrained optimization problems is the limited memory BFGS algorithm, or LBFGS, which we have recently incorporated into OPT++. In this method, information from only a subset of M Hessian columns is kept at any iteration, with $M << N$. We have found that a value of $M \approx 30$ is adequate for proteins with a few thousand atoms.

Figure 15 contains the code to minimize AMBER for a 593 atom protein known as 1e0m. The protein is encoded in pdb format, handled by our `PDB.h` library [22]. The user function `amberInit()` returns the starting point for the optimization, and initializes global variables used by `amberEval()` to perform the energy computations.

In the main routine, we first load the pdb file for this protein, and initalize the problem dimension to be three times the number of atoms. The energy function provides first order derivatives, hence we construct a nonlinear problem of type NLF1, which we then pass to the constructor of the OptLBFGS optimization object. Before running the optimization, we adjust a few parameters with values more adequate for this large-size problem. After the optimization call, the final status of the algorithm is appended to the iteration output file. The second argument to `printStatus()` avoids printing the final point (a very long vector). Instead, we obtain the final point from the nlp object and save it in PDB format.

```
#include "OptLBFGS.h"
#include "PDB.h"
USERINITFCN amberInit();
USEREVALFCN amberEval();
PDB protein;

int main() {

  protein.load("1e0m.pdb");
  int natoms = protein.nofAtoms();
  int ndim = 3 * natoms;

  NLF1 nlp(ndim, amberEval, amberInit);
  OptLBFGS optobj(ndim, nlp);

  // adjust parameters
  optobj.setMem(30);
  optobj.setMaxIter(10000);
  optobj.setFcnTol(1e-6);

  optobj.optimize();       // optimization call

  optobj.printStatus("status", false);
  ColumnVector X = nlp.getXc();
  protein.writepdb("final.pdb", X.Store() );
}
```

Figure 15: Protein energy minimization code

In Figure 16 we show a plot of the value of the energy after each of the 7224 iterations of the optimization algorithm.

The initial and final configurations for the protein are shown in figures 17. Note how minimizing the AMBER energy has effectively put the protein in a more compact configuration.

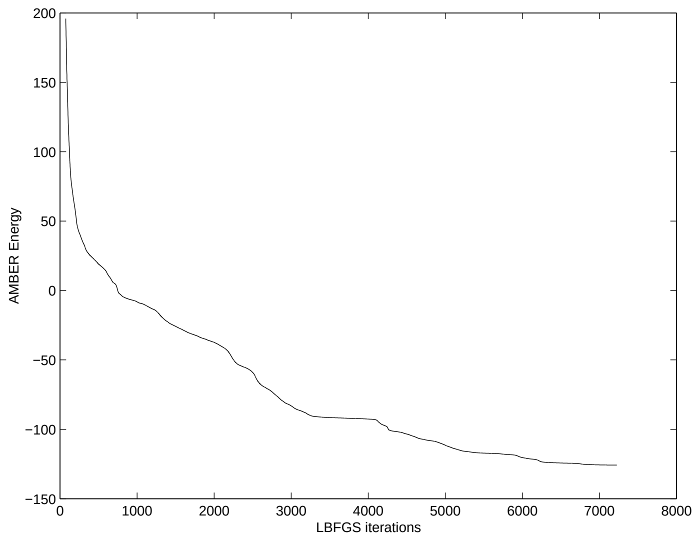


Figure 16: AMBER energy per LBFGS iteration.

5 Summary

In this report, we have presented a C++ class library for nonlinear optimization. We have proposed that a clear distinction be made between nonlinear problems and optimization methods. Based on this distinction, we have implemented a set of object-oriented classes specifically suited to each case. In this way, we have been able to develop a set of classes that address the important issues for both the users and the developers of optimization algorithms. From the point of view of a user requiring an optimization algorithm to solve a particular problem, these libraries have been written so that they are easily used. From the point of view of someone developing optimization algorithms, these classes have been designed so that new algorithms can be easily incorporated into the existing framework.

We have several methods implemented for the three main classes of optimization methods that we have described: 1) direct search methods, 2) conjugate gradient like methods, and 3) Newton like methods. Future work will concentrate on incorporating new algorithms. We are currently working on implementing new classes for large-scale optimization. Since most of the popular methods for large-scale optimization use variations of one of the methods already implemented, the extension to large-scale problems should be straightforward. Finally, we note that the libraries presented in this article should not be considered as a finished product. The true test will be the usefulness of these class libraries for solving real-world applications.

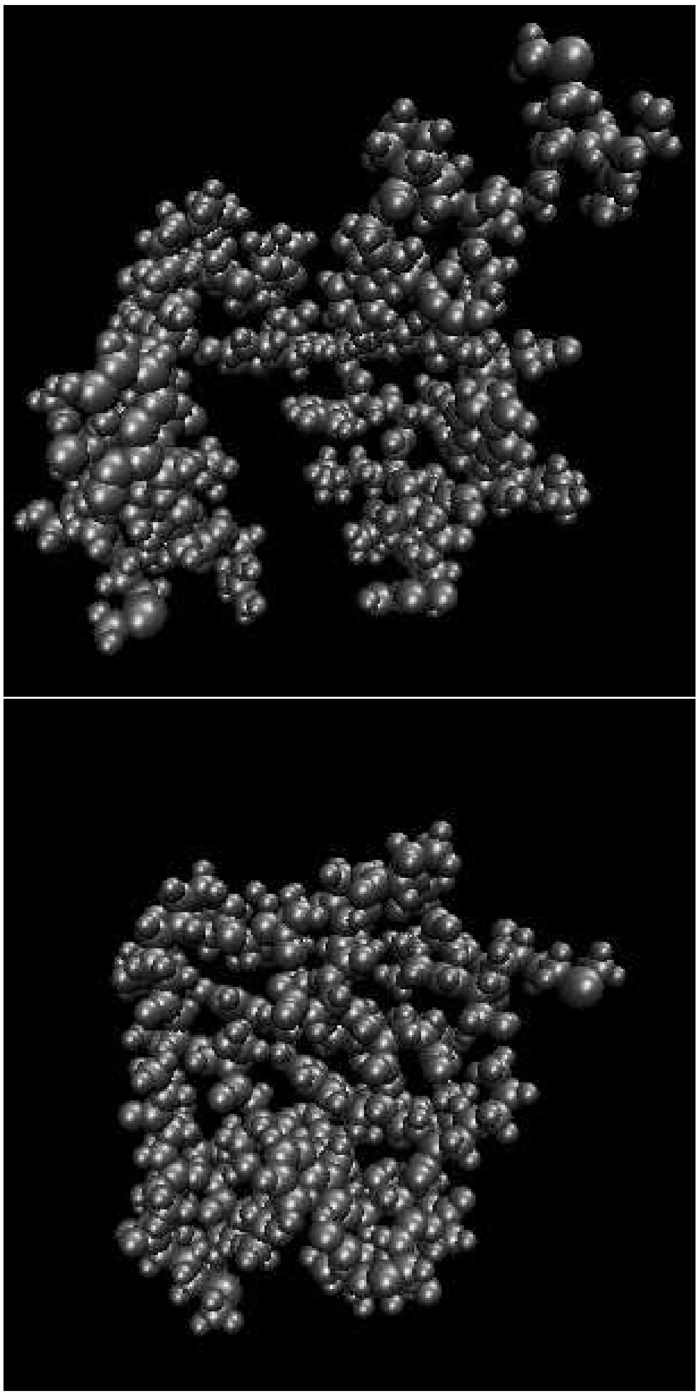


Figure 17: Protein initial and final configurations.

References

[1] Brett M. Averick and Jorge J. More. User guide for the MINPACK-2 test problem collection. Technical Report ANL/MCS-TM-157, Argonne National Laboratory, 1991.

[2] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1991.

[3] David M. Butler. Fundamentals of object-oriented programming. Limit Point Systems, Fremont CA, 1992.

[4] R. H. Byrd, R. B. Schnabel, and G. A. Shultz. Parallel quasi-newton methods for unconstrained optimization. *Mathematical Programming*, 42:273–306, 1988.

[5] R. B. Davies. NEWMAT, C++ Matrix Library - a short introduction. http://www.robertnz.net/nmintro.htm, 2003.

[6] J. E. Dennis, Jr. and V. Torczon. Direct search methods on parallel machines. *SIAM J. Opt.*, 1(4):448–474, 1991.

[7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Massachusetts, 1994.

[8] Hock, W., Schittkowski, K. (1981): Test Examples for Nonlinear Programming Codes, Lecture Notes in Economics and Mathematical Systems,Vol. 187, Springer-Verlag

[9] M. S. Gockenbach, M. J. Petro, and W. W. Symes. C++ Classes for Linking Optimization with Complex Simulations. *ACM Trans. Math. Soft.*, 25(2), pp 191-212, 1999.

[10] Allen I. Holub. *C+ C++ Programming With Objects in C and C++.* McGraw-Hill, New York, NY, 1992.

[11] P. D. Hough and J. C. Meza. A class of trust-region methods for parallel optimization. *SIAM J. Optim.*, 13(1):264–282, 2002.

[12] V. E. Howle, S. M. Shontz, and P. D. Hough. Some Parallel Extensions to Optimization Methods in OPT++. Sandia National Laboratories Technical Report SAND2000-8877, October 2000.

[13] J.C. Meza and R.A. Oliva. An object oriented library to manage the collection of Schittkowski test problems for nonlinear optimization. Lawrence Berkeley National Laboratory Technical Report LBNL-53685, 2003

[14] J.C. Meza. OPT++: An Object Oriented Class Library for Non-linear Optimization. Sandia National Laboratories, Technical Report 94-8225, 1994

[15] J.C. Meza, R.S. Judson, T.R. Faulkner, and A.M. Treasurywala, A Comparison of a Direct Search Method and a Genetic Algorithm for Conformational Searching. Journal of Computational Chemistry, Vol. 17, No. 9, pp. 1142-1151, 1996.

[16] J.C. Meza and M.L. Martinez. Direct Search Methods for the Molecular Conformation Problem. Journal of Computational Chemistry, Vol. 15, No. 6, pp. 627-632, 1993.

[17] J.C. Meza and T.D. Plantenga. Optimal Control of a CVD Reactor for Prescribed Temperature Behavior. Sandia National Laboratories, Technical Report 95-8222, 1995.

[18] C.D. Moen, P.A. Spence and J.C. Meza. Optimal Heat Transfer Design of Chemical Vapor Deposition Reactors. Sandia National Laboratories, Technical Report 95-8223, 1995.

[19] Jorge J. More and David J. Thuente. Line search algorithms with guaranteed sufficient decrease. Technical Report MCS-P330-1092, Argonne National Laboratory, 1992.

[20] Jorge J. More and Stephen J. Wright. *Optimization Software Guide.* SIAM Press, Philadelphia, PA, 1993.

[21] Dave Nichols, Geoff Dunbar, and Jon Claerbout. The C++ language in physical science. In *OON-SKI '93*, pages 339–353, April 1993. Proceedings of the First Annual Object-Oriented Numerics Conference.

[22] R.A. Oliva. An Object-Oriented Library for Molecular Dynamics Energy Computations. Lawrence Berkeley National Laboratory Technical Report LBNL-XXXXX, 2003

[23] E. M Gertz and S. J. Wright. Object-Oriented Software for Quadratic Programming. *ACM Trans. Math. Soft.*, 29(1), pp 58-81, 2003.

[24] K. Schittkowski. More Test Examples for Nonlinear Program-ming. Lecture Notes in Economics and Mathematical Systems, Vol. 282, Springer-Verlag, 1987.

[25] K. Schittkowski Test Examples for Nonlinear Programming User's Guide, Online at http://www.klaus-schittkowski.de, 2002

[26] Ronald Schoenberg. An object-oriented design of an optimization module. In *OON-SKI '93*, pages 132–139, April 1993. Proceedings of the First Annual Object-Oriented Numerics Conference.

[27] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1987.

[28] S. J. Benson, L. Curfman McInnes, and J. J. More. A Case Study in the Performance and Scalability of Optimization Algorithms. *ACM Trans. Math. Soft.*, 27(3), pp 361-376, 2001.